Online Load Profiling and Dynamic Optimization in the Java™ Hotspot™ VM

Andrew Trick and Xiaoyi Guo

Hewlett-Packard Company

Ember Project: Optimizing Code Generator for Itanium®

Load Profiling Contributors:
Andrew Trick
Xiaoyi Guo
Richard Allen
Laurent Morichetti
Swaroop Dutta

Disclosure: Patents have been initiated for some ideas contained in this presentation.

Outline

- Motivation and overview
- PMU sampling overhead
- Context-sensitive interpretation of PMU data
- Optimizations driven by load profile
- Future work

SPECjbb2005

```
Processor type:      Itanium2 16K L1, 256K L2, 9M L3
Processor speed:     1600 MHz

----------------------BE Components-------------------
Unstall     BE     Score  --------Daccess-------   RSE
Execute    Flush   Board  L1Dtlb  L2Dtlb  Dcache  Active
------------------------------------------------------
 44.84%    4.10%   0.32%   0.44%   0.08%  45.29%  0.74%

 --Latency buckets as % Misses-
  L2   --L3-- ------Memory------
   7   14   64   150 250 350 450
 ----------------------------
  34% 45% 18%   1%   1%   2%   0
```

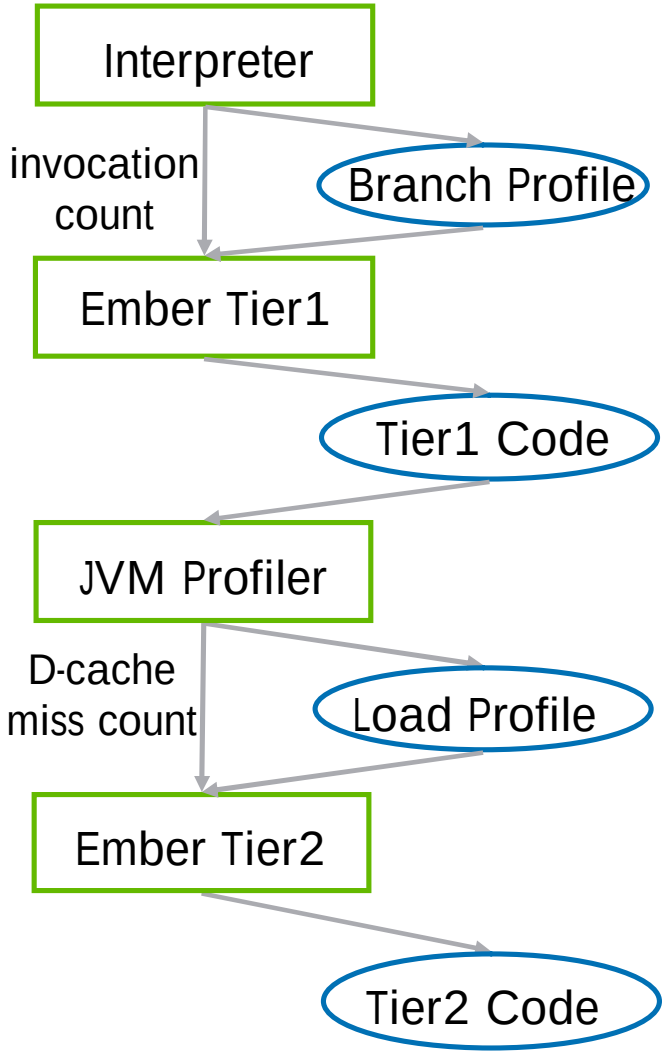
SPECjAppServer2004

```
Processor type:      Itanium2 16K L1, 256K L2, 9M L3
Processor speed:     1600 MHz

----------------------BE Components-------------------
Unstall     BE     Score  --------Daccess-------   RSE
Execute    Flush   Board  L1Dtlb  L2Dtlb  Dcache  Active
------------------------------------------------------
 28.70%    4.06%   0.21%   0.61%   0.97%  51.53%  3.42%

 --Latency buckets as % Misses-
  L2   --L3--  ------Memory------
   7   14   64   150 250 350 450
 ----------------------------
  56% 28% 10%   3%   2%   1%   0
```

Load Profiling Overview

- Target
  - Hotspot Java Virtual Machine
  - Dynamic compiler
- Method
  - Use Itanium Performance Monitoring Unit (PMU)
    - Relinquish PMU after recompilation
  - Online profiling – no training run

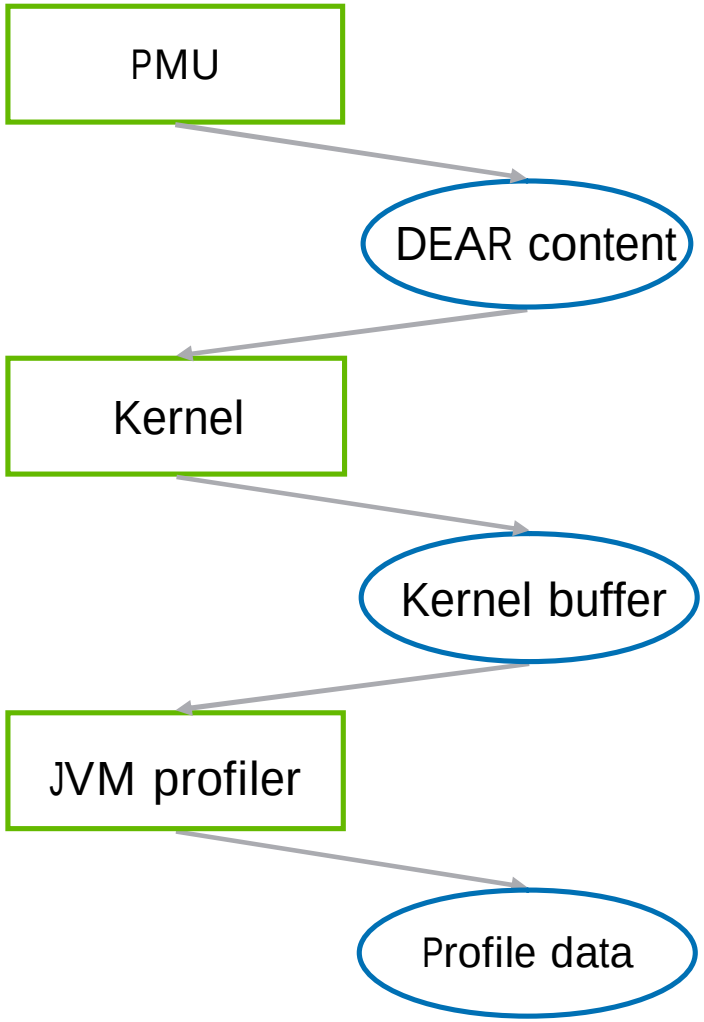
Two Levels of Compilation



- Tier 1
  - Triggered by repetitive interpretation
  - Fast compilation
  - Moderate branch profile driven Inlining
- Tier 2
  - Triggered by frequent PMU event
  - More expensive optimizations
  - Aggressive Inlining
  - Optimizations sensitive to load profile

PMU Sampling Overhead

Reducing Sampling Overhead



- L1 miss captured in Data Event Address Register (DEAR)
- PMU interrupts kernel every N DATA_EAR_EVENTS
  - Samples the DEAR
  - Appends to buffer
- JVM signaled on full kernel buffer
  - User-level signal expensive
  - Buffering provides performance optimizations
    - Processing samples in blocks
    - Prefetching blocks of data

D-cache Miss Profiling Overhead

- 1.5% CPU overhead during collection
- Most attributable to kernel interrupt handling
- Configuration used
  - sampling rate = 1000 DATA_EAR_EVENTS
    - In reality, one sample per 8000 loads that miss L1
  - buffer size = 1000 samples
    - 8% CPU overhead without buffering

Collecting D-cache Miss Profile

- For D-cache miss event, PMU provides:

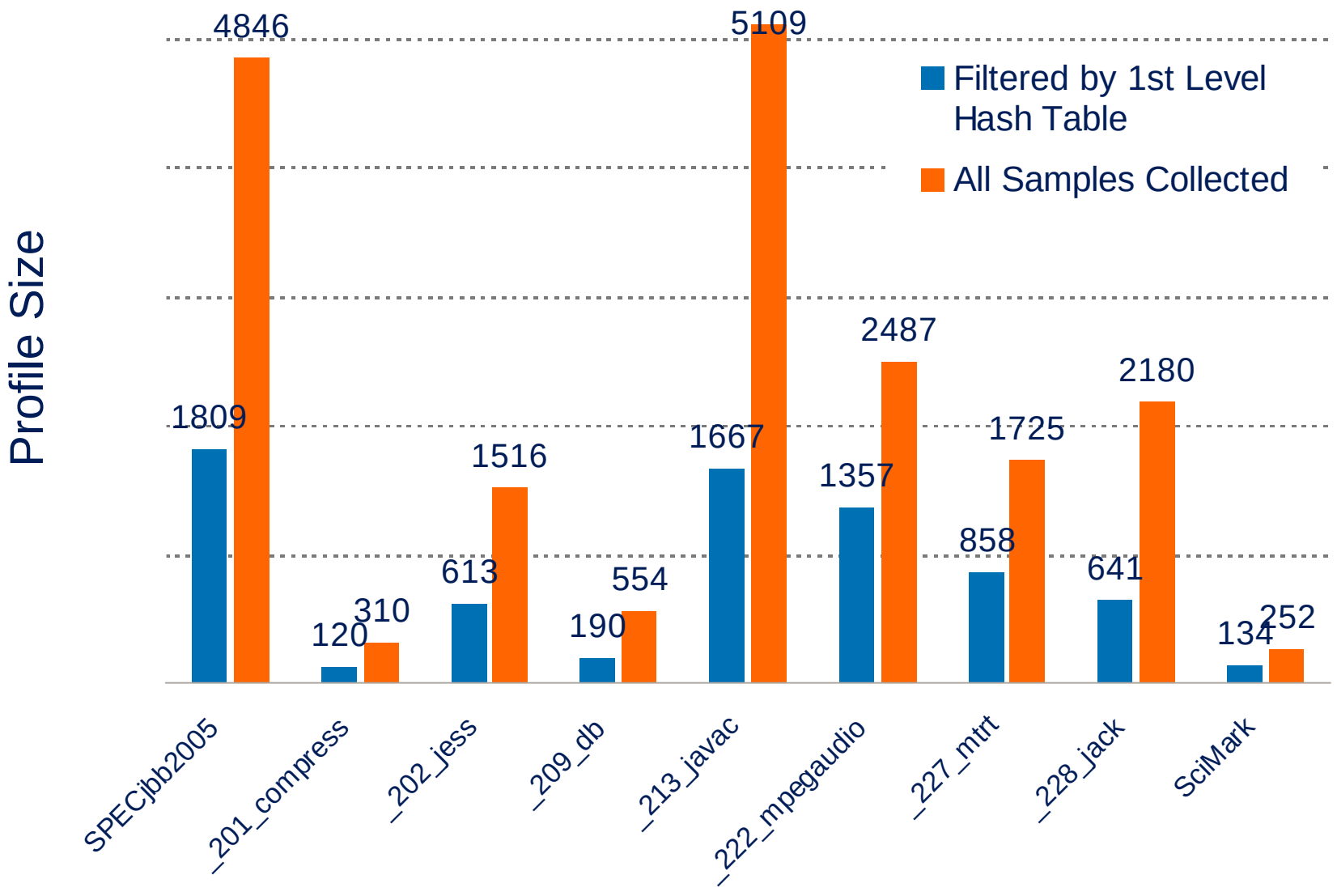
- Online profiler
  - Aggregates data for IP
  - Stores in hash table
- Hash table data format:

- Average latency can be computed at any time

Two Profile Data Hash Tables

- First Level
  - Highly tuned, fixed sized
  - Aggregates buffer samples into profile records
  - Accessed only by profiler thread, no synchronization
- Second Level
  - Profiler thread scans first level table every N buffers
    - Promote records with high miss frequency, discard others
  - After promotion, method may be marked for recompilation
  - Second-level table is read by the compiler thread

Profile Noise Reduction



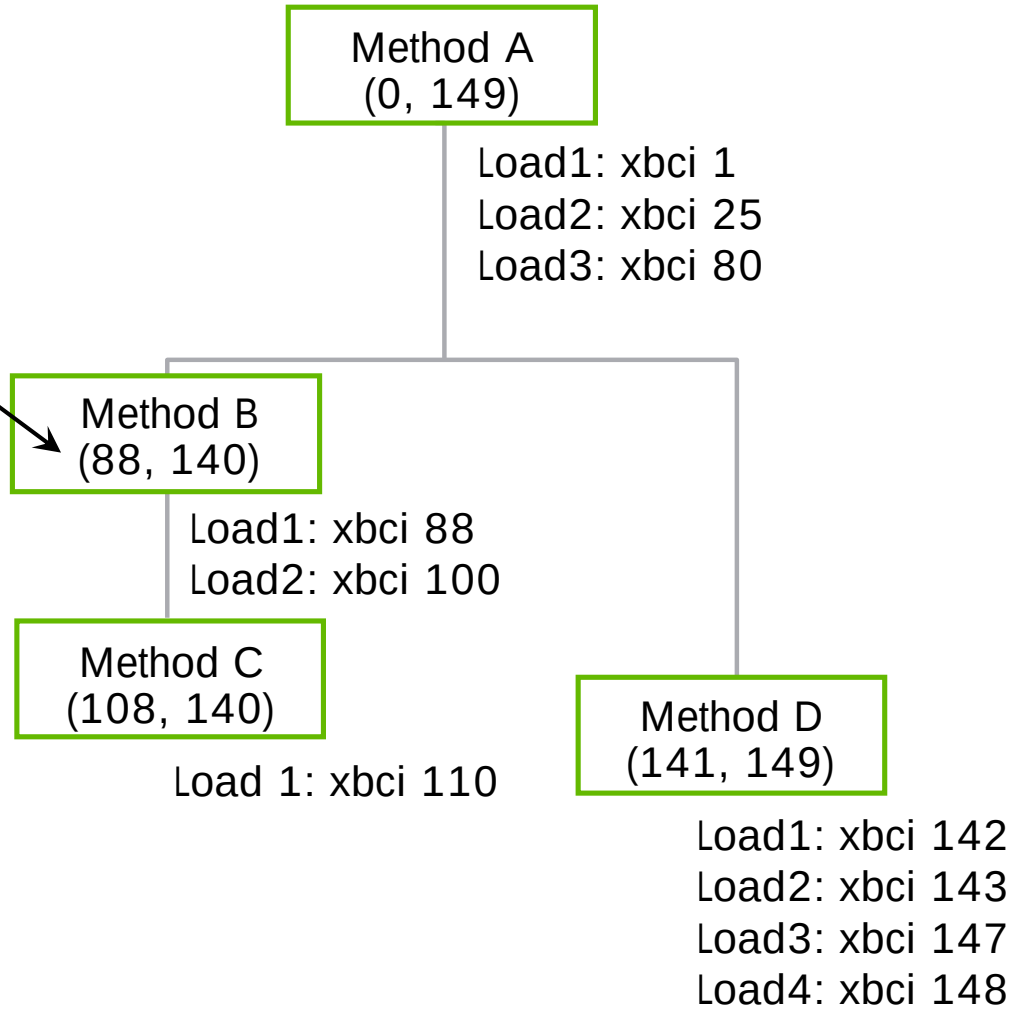
Context-Sensitive Interpretation of PMU Data

Profile Data Correlation

- Annotate Tier1 code
  - Map IP to “load descriptor”
    1. Java bytecode index (BCI) of the load
    2. Inlined call-chain & BCI of each call
- Profile identifies a load by its IP in Tier 1 code
- During Tier 2 compilation
  - For each load, find longest matching call chain in Tier 1
  - Use the Tier 1 IP to lookup profile data in hash table

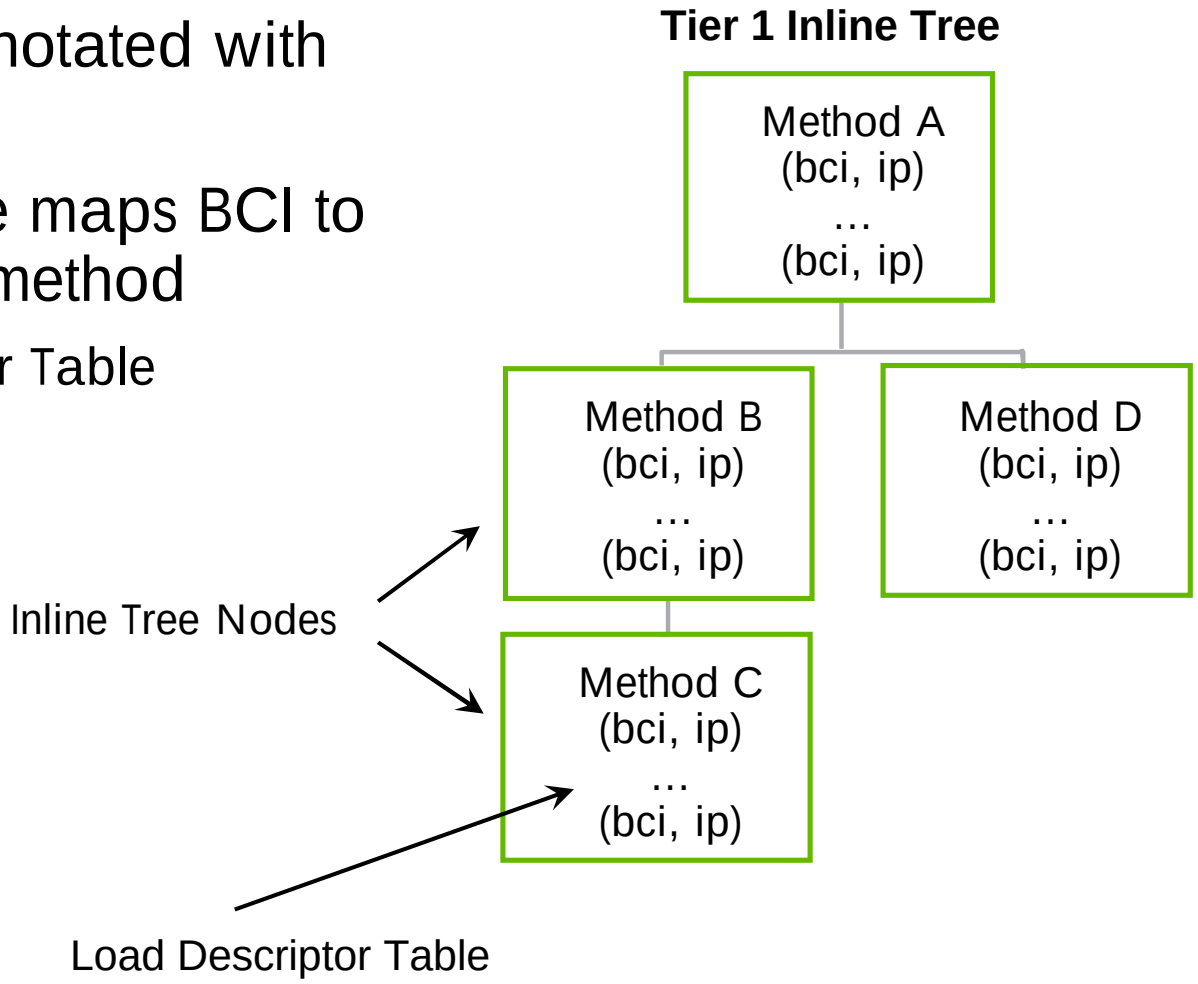
Tier 1: Recover Call Chain from xbci

- Load instruction tagged with Extended BCI (xbci)
  - Unique identifier across compilation scope
- Associate nodes in inline tree with xbci range
- Antecedent-descendant relationships: Queried by comparing xbci values
- Call chain corresponding to any xbci value can be computed efficiently



Tier 1: Load Descriptor Annotation

- Tier 1 code annotated with inline tree
- Inline tree node maps BCI to load IP in that method
  - Load Descriptor Table

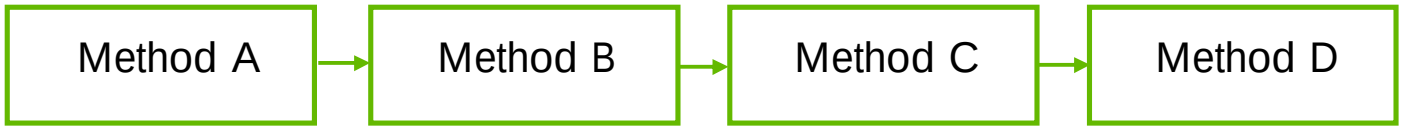


Tier 2: Match Inlined Call Chain

- Find longest Tier 1 compiled call chain that matches current Tier 2 inlined call chain
  - Provides most accurate load profile data
- Algorithm guarantees only one amortized lookup of Tier 1 compiled method per inlined call
  - Once we find the best match for a node in Tier 2 inline tree we cache the result
  - No repetitive search of Tier 1 methods' inline trees

Call Chain Matching Example

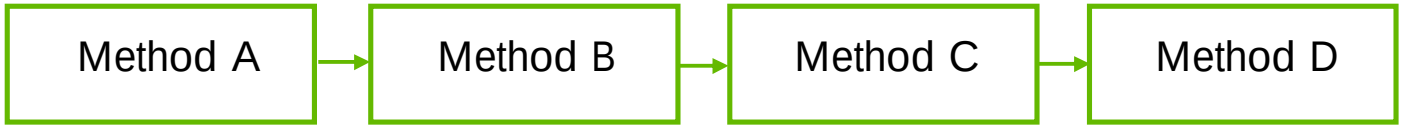
Tier2 Call Chain:



Need to find Tier1 call chain match for all nodes on Tier2 call chain

Call Chain Matching Example

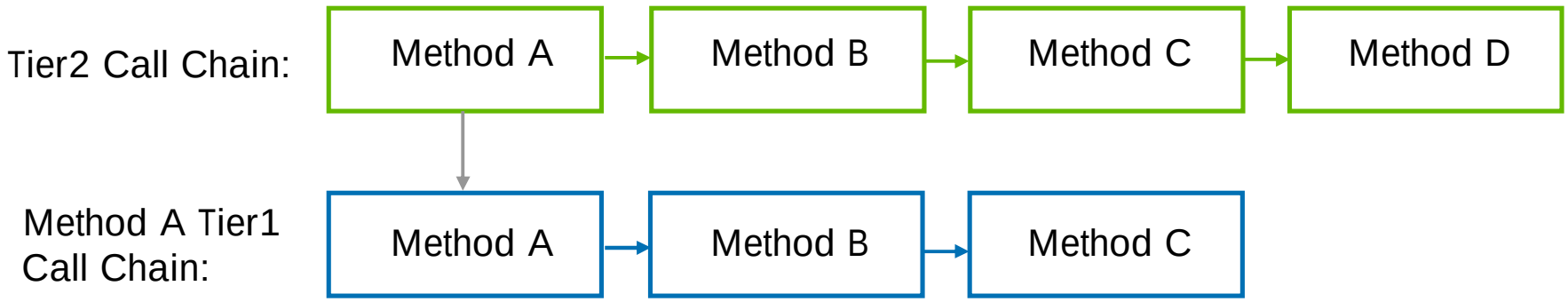
Tier2 Call Chain:



Method A Tier1 Call Chain:

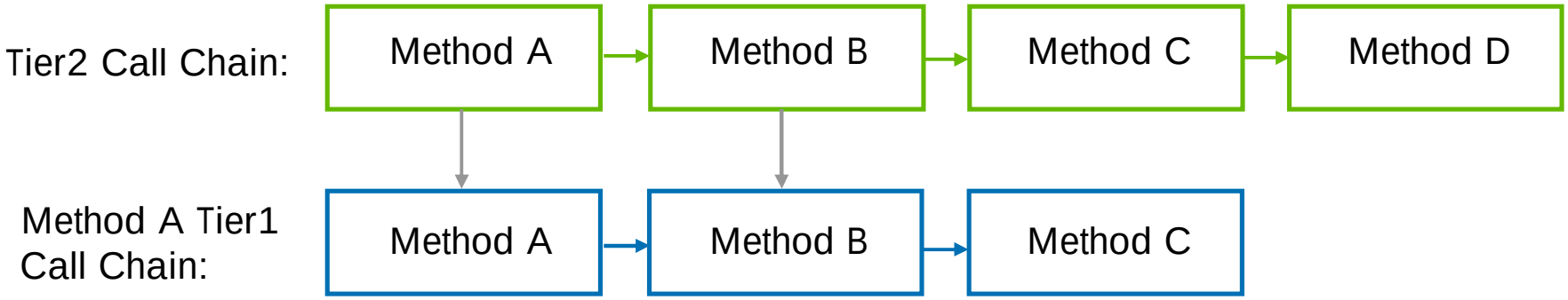
Start with method A's Tier1 inline tree

Call Chain Matching Example



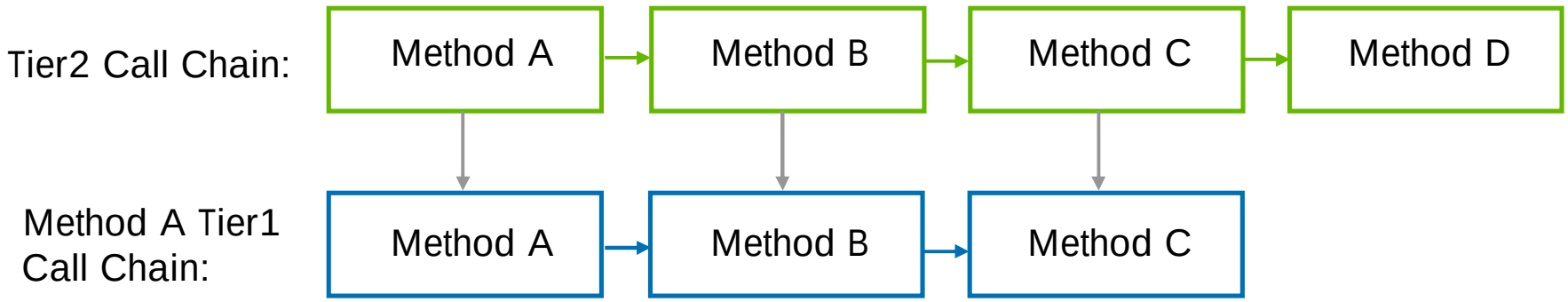
Found match for method A

Call Chain Matching Example



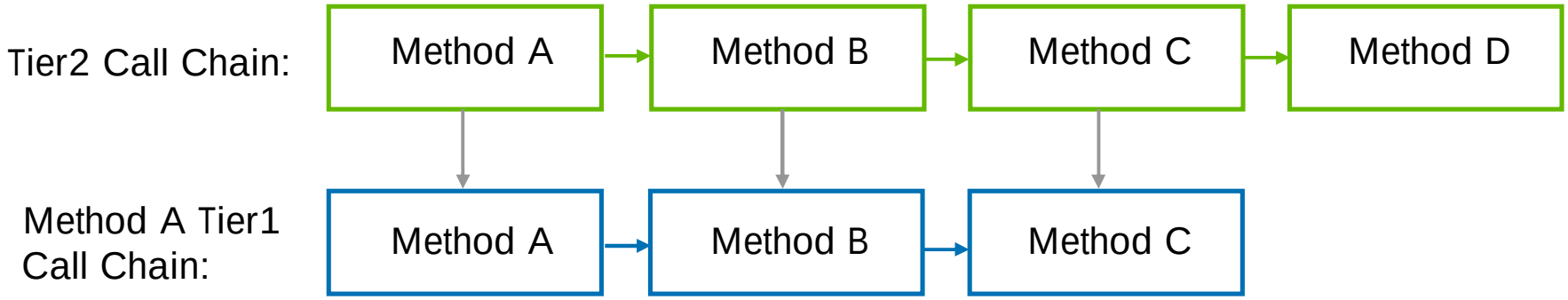
Found match for method B

Call Chain Matching Example



Found match for method C

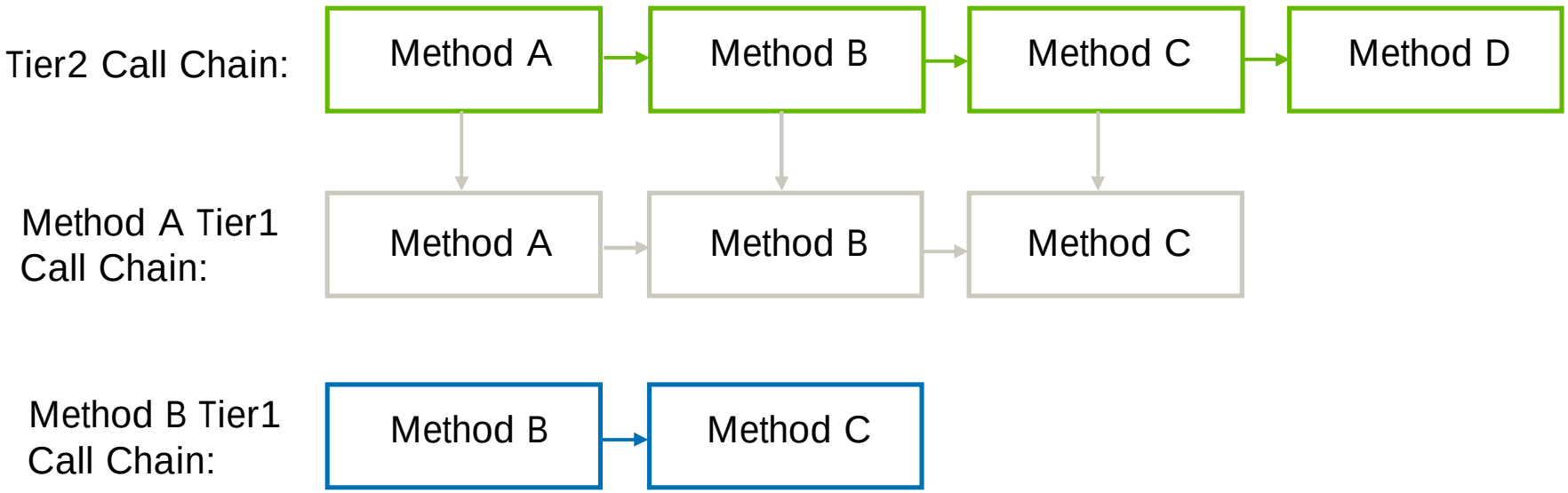
Call Chain Matching Example



In method A's Tier1 inline tree, look for D in method C's children

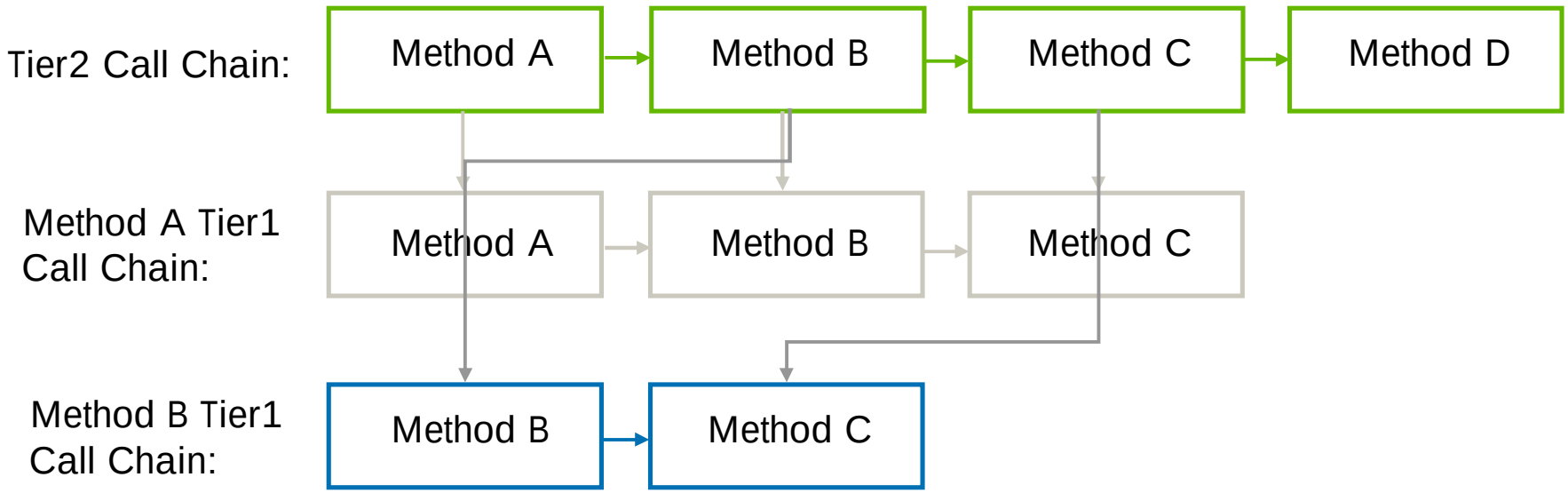
Not found

Call Chain Matching Example



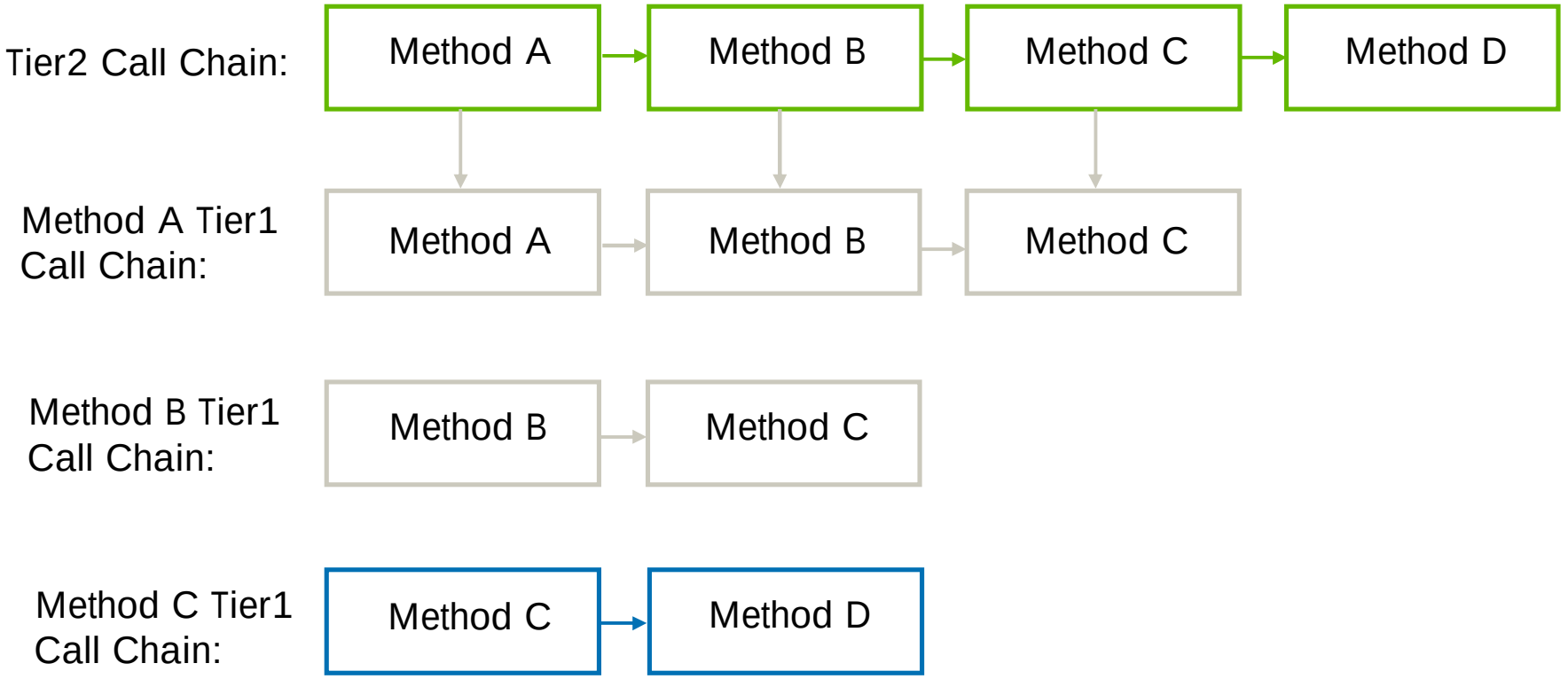
Now go to method B's Tier1 inline tree, look for call chain B -> C -> D

Call Chain Matching Example



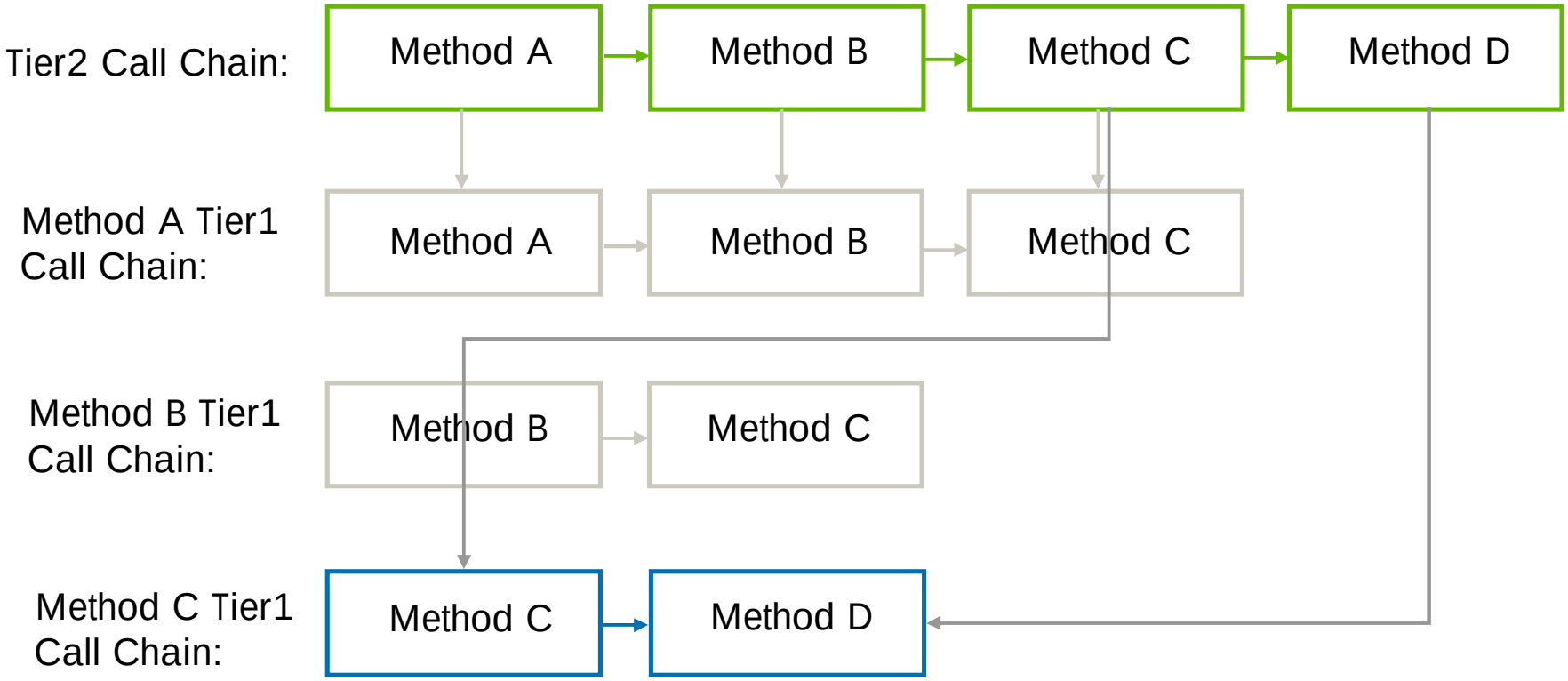
Cannot find call chain B -> C -> D

Call Chain Matching Example



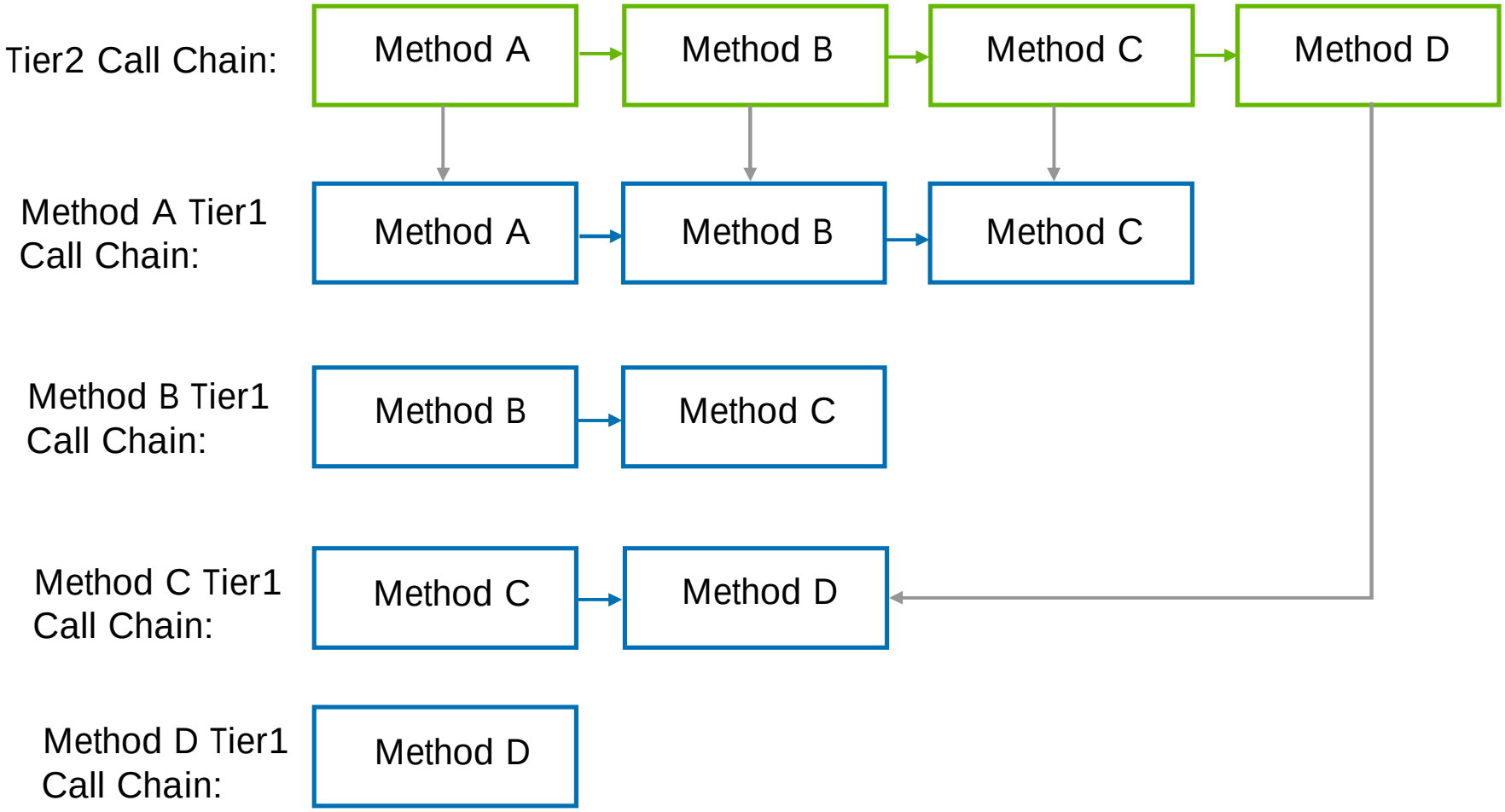
Now go to method C's Tier1 inline tree, look for call chain C -> D

Call Chain Matching Example



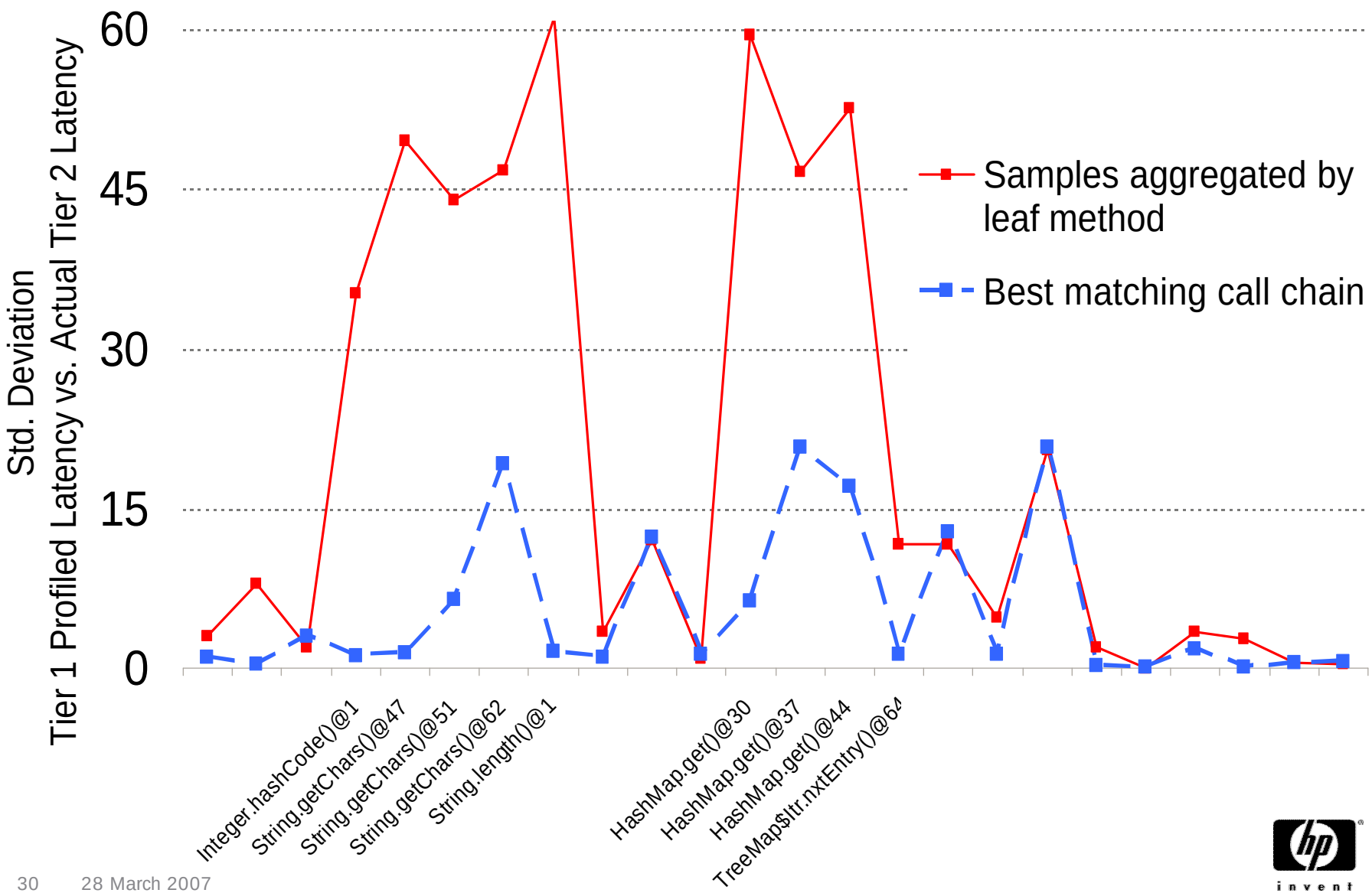
Found call chain C -> D

Call Chain Matching Example



Match result for all nodes on Tier 2 call chain

Context Sensitivity Improves Accuracy



invent

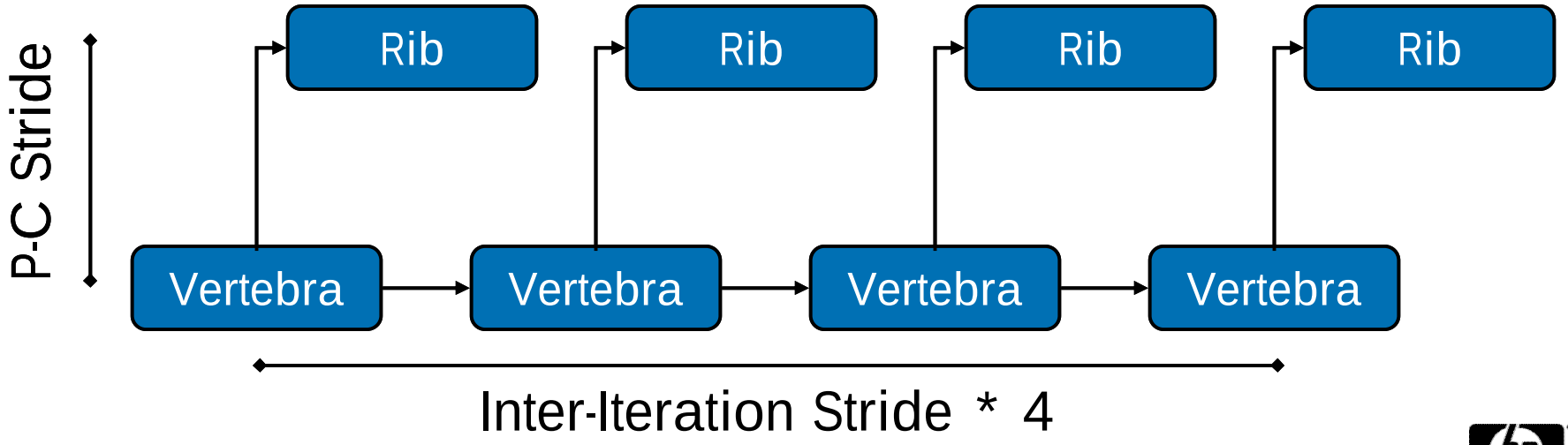
Load Profiling Summary

- Low overhead
  - Compilation time < 0.7%
  - PMU data collection <2%
  - Code cache size increase < 8%
    - Tier 1 inline tree & load descriptors
    - Tier 2 compiled methods
  - Profile data size is ~ 3% of code cache size
- Context-sensitive profile
  - Lookup is simple, cheap
  - Improves accuracy

Optimizations Driven by Load Profiling

Stride Prefetching: As good as it gets

- Array prefetching (problem solved, still suffering)
- Prefetching linked data structures (feel lucky?)
  - Inter-iteration, same load
  - Intra-iteration, parent-child relationship



Stride Profiling

- Current low-overhead load profiler
  - Does not record data address—potentially much more expensive
  - Does indicate which loads may be worth stride profiling
- Instrumentation
  - Temporarily activate call to profile stub to record data address
  - Compiler can easily discover the “parent” load and pass both data addresses
  - Can detect regular parent-child offsets
    - e.g. String->char[]

Stride Profiling

- PMU sampling of data addresses
  - Interrupt on every sample
  - Still no consecutive addresses
    - PMU ignores all L1 hits
    - PMU randomly drops 7/8 misses
  - Attempt to deduce stride using GCD
    - See Intel's Ispike paper
  - Detects only inter-iteration, same load strides

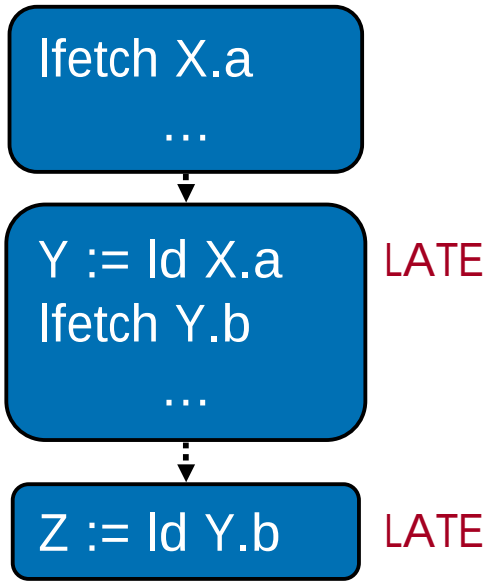
SPECjbb2005 Stride Profile

- ~10% of “delinquent” loads exhibit a dominant non-zero constant stride across iterations > 80% of the time
  - Many delinquent loads have zero stride
    - Loads to evicted static fields
    - No method to stride prefetch
- ~10% of delinquent loads are at constant stride from their “parent” load
- Constant strides found both before and after GC

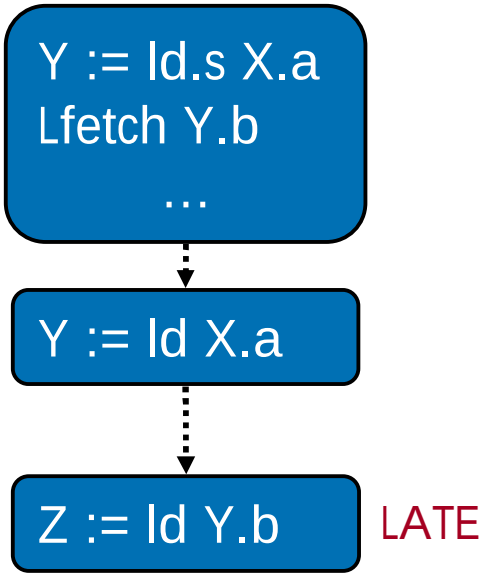
Straight-Line Prefetching

- Profile-Driven Prefetching Prototype
  - Insert lfetch instructions prior to scheduling
  - Can prefetch across arbitrary CFG regions
  - From multiple address sources
- Examples (LATE = identified delinquent by profiler):

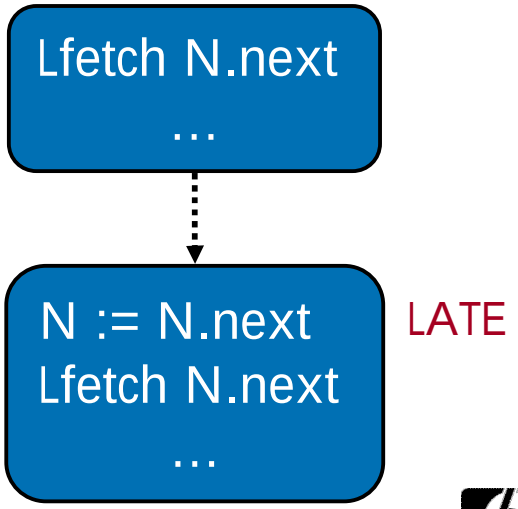
Direct Prefetch
Chained Loads



Indirect Prefetch



Recurrent Prefetch



Prefetching Pitfalls

- 10% degradation with naïve heuristics
- Schedule length increases, but not a major factor
  - Replacing lfetch with nops fixes degradation
- Can stall at prefetch if address unavailable
- Too many outstanding misses clogs the memory subsystem
  - To the same or different addresses
  - Prefetch candidate threshold can be set much lower on Montecito before degrading performance

Initial Prefetching Heuristics

- Add heuristics to avoid useless prefetches
  - From -10% to +2% on jbb2005
- Approaches
  - Consider branch frequency
  - Consider dynamic cycles
    - Time between prefetch & original load
    - Esp. look for other delinquent load to different object
  - Barrier
    - Block containing a missing load to the same object
  - Relative miss ratio
    - Helps determine whether a load misses frequently relative to it's parent, before speculating the parent

Prefetching Improvements

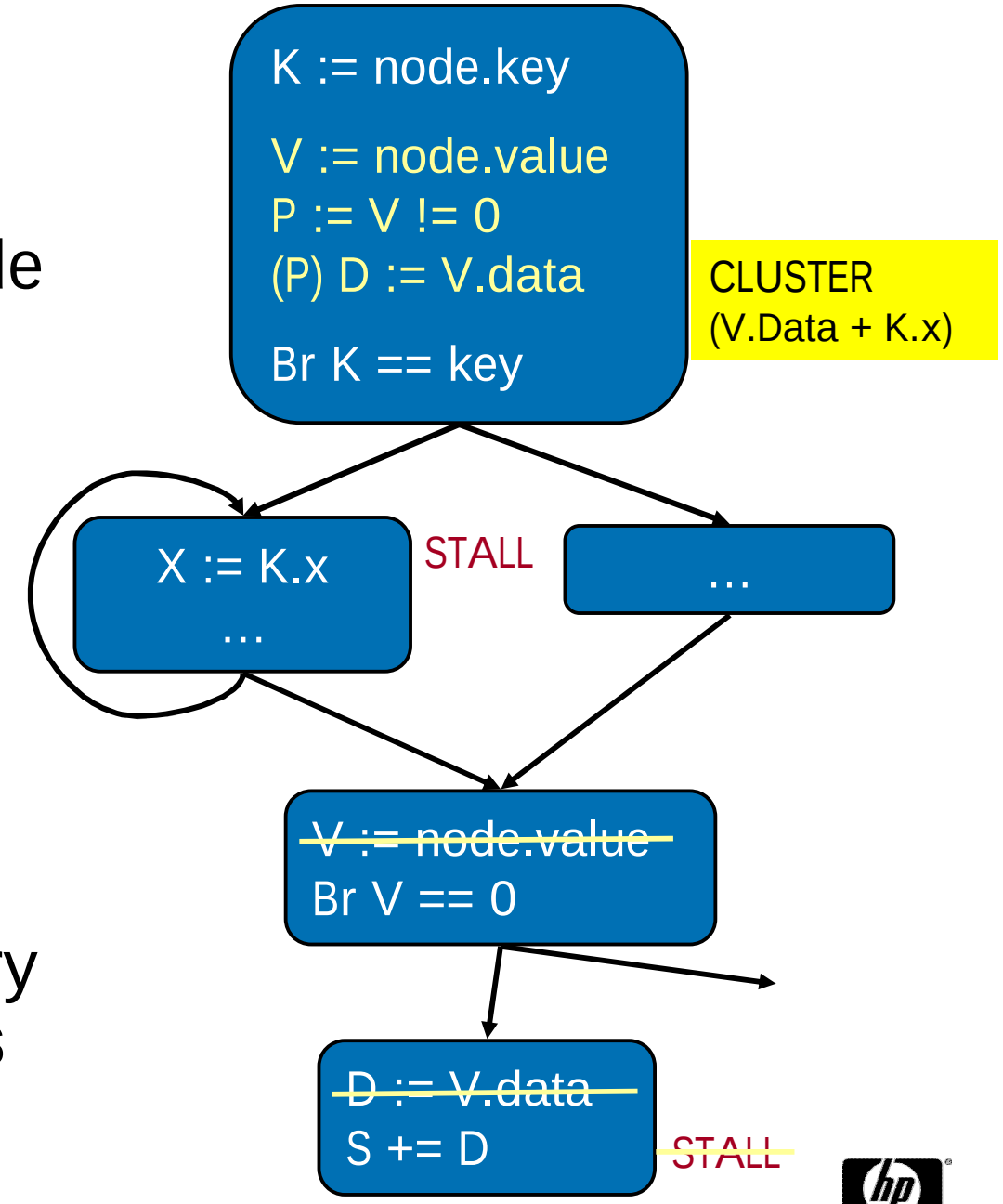
- Absolute miss ratio
  - Helpful for determining profitability
  - Very difficult to deduce
- Heuristics need work
  - Still performance opportunities
  - Initial algorithm very simple & efficient
- Scheduler should be made prefetch-aware

Load hoisting in Java

- Safely hoist loads from Java heap
  - Valid for loads to constant offset
    - True for any non-array load and for peeled array loads
    - Pad the end of your heap
  - Ignore type check/cast
  - Ignore bounds check for constant index
- Still need a null check
  - Could make page zero readable but impacts supportability
  - No H/W + OS support for non-faulting page zero load without recovery code

Predicated GCM (The Good)

- Leverage existing code motion framework
  - GCM is cheap
  - No new phases
  - No new heuristics
- Not limited by the scope of scheduling regions
- No additional memory ops on frequent paths



Predicated GCM (The Bad)

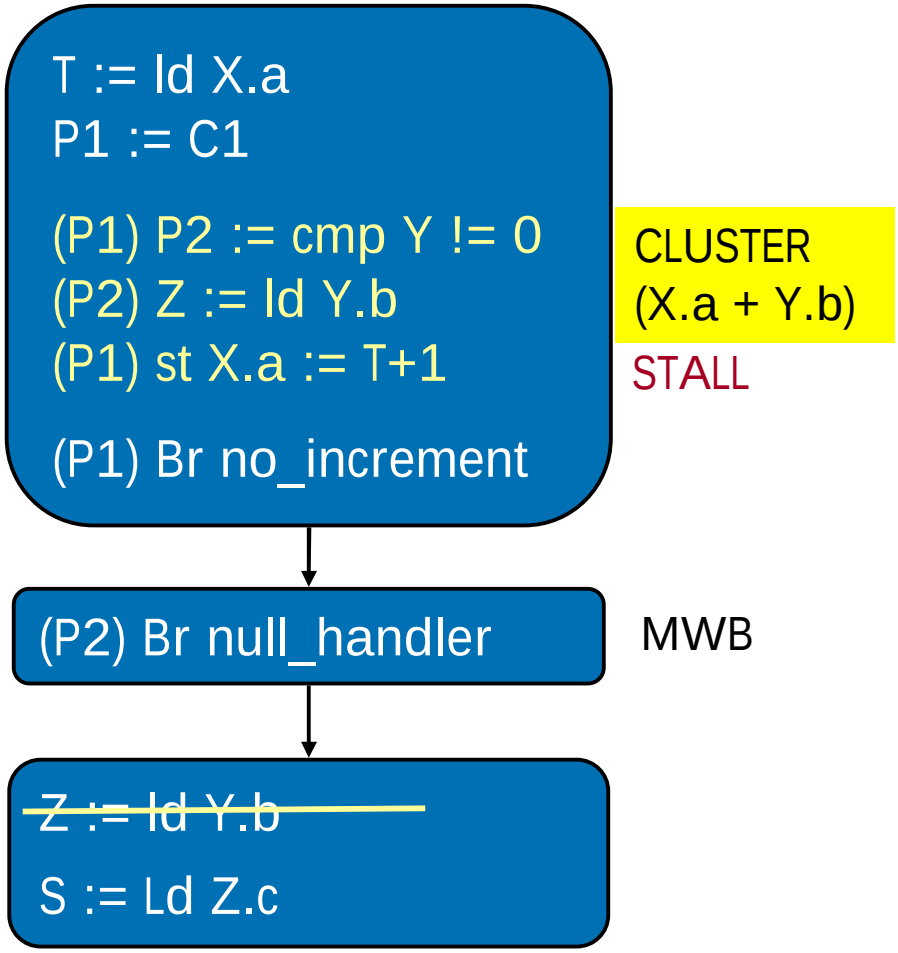
- Restricted by potential memory interferences, including calls
- Restricted to dominator tree
  - Can’t clone load into multiple paths
  - Can’t help recurrent loads
- Load profile does not yet surpass static heuristics
  - Important to hoist selectively
  - Current implementation limited by type checks

Profile-Sensitive Scheduling

- Early time still uses unit latency
- Dependence height (for priority among ready instructions) includes profiled load latency
  - Capped at some reasonable value
  - Scaled by relative miss ratio
- Dependence height of consumers of delinquent loads adjusted downward on-the-fly
  - Only if they are ready too early
  - Adjusted_height -= load_ready_time – current_cycle

Predicated Hoisting

- Hoist out of home block if fully-resolved predicate (FRP) is available
- In theory, almost as good as hardware control speculation w/ recovery
- But FRP restriction can limit code motion



Future Work

- Prefetch-friendly garbage collection and allocation
- Profile BTB
  - Execution frequency concurrent with miss frequency
    - Deduce miss ratio
  - ~20% of execution spent in methods that do not experience frequent d-cache misses
- Integrated load-hoisting & prefetching phase
- Prefetch-aware scheduling

References

- Dynamic Profile-Guided Optimization in the BEA Jrockit™ JVM
  - Aundhe, Eastman, Kasten, and Knight. 2005.
- Prefetch Injection Based on Hardware Monitoring and Object Metadata.
  - Adl-Tabatabai, Hudson, Serrano, and Subramoney. 2004.
- Simple and Effective Array Prefetching in Java.
  - Cahoon and McKinley. 2002.
- Efficient Discovery of Regular Stride Patterns in Irregular Programs and Its Use in Compiler Prefetching.
  - Wu. 2002.
- Ispike: A Post-link Optimizer for the Intel Itanium Architecture.
  - Luk, Muth, Patil, Cohn, Lowney. 2004.

Questions